Final Project Phase 4 Turn-in - Data Generation Process Documentation

Team members: Thomas, Chris, and Sarah

Brief Overview

Our data is generated through a C# console application that requires very minimal user interaction and can insert large quantities of data directly into the database using Entity Framework Core. The amount of data inserted is configurable by changing numbers in the code in a few key spots.

Below are explanations of the different files in our application, along with code snippets.

Program.cs

The Program.cs file sets up the application to work with Entity Framework Core for easy integration with the database. It also handles any exceptions in setting up EF Core by printing the exception messages to the console.

Program.cs also contains the logic that triggers data generation at the press of any key. The GenerateData method called near the end of the file is where the number of iterations (days to insert flights for) and the number of products (concessions purchased by passengers per day) can be inputted.

Code in Program.cs file

```
// Main driver for console creation of queries to insert into
scheduled_flight
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;
using AirportFlightScheduler.Data;

namespace AirportFlightScheduler;

public class Program
{
    static async Task Main(string[] args)
    {
        DbContextOptions<AirlineContext> options;

        //Initialize DbContextOptions object w/ user secret
        try
        {
            var configuration = new
```

```
ConfigurationBuilder().AddUserSecrets<Program>().Build();
            var connectionString =
configuration.GetConnectionString("DefaultConnection");
            options = new
DbContextOptionsBuilder<AirlineContext>().UseNpgsql(connectionString).Optio
ns;
            Console.WriteLine($"DBContext successfully created new options
with connection string: \n{connectionString}\n");
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.ToString());
            return;
        }

        Console.WriteLine("Press any key to start data generation...");
        Console.ReadLine();

        Console.WriteLine("Generating data...");
        FlightDataGenerator generator = new(options);
        await generator.GenerateData(15, 3);
        Console.WriteLine("Done.");
    }
}
```

EF Core Classes

As part of the functionality of Entity Framework Core, classes were generated for each of our database tables so that they could be used for model binding to table rows. EF Core also generated a DbContext class called AirlineContext.cs, which is used in integrating with the database.

FlightDataGenerator.cs

GenerateData and GenerateSingleDay are the two main methods that start both of the data generation chains. GenerateData loops through a given amount of iterations or days and runs GenerateSingleDay for each iteration. GenerateSingleDay runs all of the data generation functions in order.

```
    // Iterate through the data generations functions a set number of times
    public async Task GenerateData(int iterations, int numProducts)
    {
        for (int i = 0; i < iterations; i++)
        {
            using (_context = new AirlineContext(_contextOptions))
            {
                await GenerateSingleDay(numProducts);
                Console.WriteLine($"Done with day: {i + 1}/{iterations}");
            }
        }
    }

    // Run all of the data generation functions
    private async Task GenerateSingleDay(int numProducts)
    {
        // Data Generation Chain 1
        // SeatType -> PlaneTypeSeatType -> PlaneType -> (Airport, Plane) ->
ScheduledFlight -> FlightHistory -> Reservation -> (Seat, Payment)
        List<int> scheduledFlightIds = await GenerateScheduledFlights();
        Console.Write("1");
        List<int> flightHistoryIds = await
GenerateFlightHistory(scheduledFlightIds);
        Console.Write("2");
        List<int> reservationIds = await
GenerateReservations(scheduledFlightIds);
        Console.Write("3");
        List<int> seatIds = await GenerateSeats(reservationIds);
        Console.Write("4");
        List<int> paymentIds = await GeneratePayments(reservationIds);
        Console.Write("5");

        // Data Generation Chain 2 (A LOT SIMPLER)
        // Reservation -> (Seat, Payment) -> ConcessionPurchase ->
ConcessionPurchaseProduct <- Product
        await GenerateConcessionPurchases(seatIds, numProducts);
        Console.Write("6\n");
    }
```

1. It starts with scheduled flights. GenerateScheduledFlights generates the max number of flights possible in a single day and then returns a list of all the generated flight ids.
2. GenerateFlightHistory uses the scheduled flight ids to create entries for each scheduled flight in the flight history table.
3. GenerateResrvations also uses the scheduled flight ids to create rows in the Reservation table. It also calls GeneratePassengers which creates passengers to make reservations on those flights. It returns a list of reservation ids.
4. GenerateSeats generates however many seats are given in the seat count column for each reservation
5. GeneratePayments makes a payment row for each reservation. This function ends the first data generation chain.

1st Generation Chain

```
    // Generate a given number of fake passengers
    private async Task<List<int>> GeneratePassengers(int passengerCount)
    // Generate one day worth of flights in the in the ScheduledFlight table
    private async Task<List<int>> GenerateScheduledFlights()
    // Generate one entry in the flight history for every entry in the
scheduled flights
    private async Task<List<int>> GenerateFlightHistory(List<int>
scheduledFlightIds)
    // Given a list of scheduledFlightIds, generate all the necessary
reservations in the reservation table
    public async Task<List<int>> GenerateReservations(List<int>
scheduledFlightIds)
    // Given a list of reservationIDs, generate all the necessary seats in the
seat table
    public async Task<List<int>> GenerateSeats(List<int> reservationIds)
    // Given a list of reservationIDs, generate all the necessary payments in
the payment table
    public async Task<List<int>> GeneratePayments(List<int> reservationIds)
```

6. The second generation chain uses a single function called GenerateConcessionPurchases that creates rows in the concession_purchase_product, concession_purchase, and the payment table.

Loop section of the GenerateConcessionPurchases function (2nd generation chain)

```
        foreach (var seat in validSeats)
        {
            // Select `numProducts` random products
            var selectedProducts = products.OrderBy(_ =>
random.Next()).Take(numProducts).ToList();

            // Calculate the total cost of the products
            decimal totalCost = selectedProducts.Sum(p => p.Price);

            // Create a Payment entry
            var payment = new Payment
            {
                Amount = totalCost,
                ReservationId = seat.ReservationId
            };
            payments.Add(payment);

            // Create a ConcessionPurchase entry and link it to the payment
            var concessionPurchase = new ConcessionPurchase
            {
                SeatId = seat.Id,
                Payment = payment // Link the Payment object directly
            };
            concessionPurchases.Add(concessionPurchase);

            // Add the ConcessionPurchaseProduct entries and link to the
ConcessionPurchase
            foreach (var product in selectedProducts)
            {
                concessionPurchaseProducts.Add(new ConcessionPurchaseProduct
                {
                    ConcessionPurchase = concessionPurchase, // Link the
ConcessionPurchase object directly
                    ProductId = product.Id,
                    Quantity = 1
                });
            }
        }
```

PlaceholderData.cs

These functions are what we used to randomize the data inserted into the tables. The flightTimes dictionary holds all of the possible flights. It uses a tuple with the departure and arrival airport as a key and the flight time as the associated value.

PlaceHolderData Methods and Flight Time Dictionary

```
    public static string GeneratePassengerName() =>
        $"passenger-{rand.Next(100000000, 1000000000)}";

    public static string GeneratePassportId() =>
        $"{rand.Next(100000000, 1000000000)}";

    public static string GeneratePhoneNumber() =>
        $"({rand.Next(100, 1000)}) {rand.Next(100, 1000)}-{rand.Next(1000,
10000)}";

    public static string GenerateEmail() =>
        $"{Guid.NewGuid().ToString("N").Substring(0,
10)}@{Guid.NewGuid().ToString("N").Substring(0, 9)}.com";

    public static string GenerateAddress()
    {
        string[] streets = { "Main St", "Maple Ave", "Elm Dr", "Oak Blvd",
"Pine Ct" };
        string[] cities = { "Springfield", "Riverton", "Hillcrest", "Lakeside",
"Fairview" };
        string[] states = { "NY", "CA", "TX", "FL", "WA" };

        return $"{rand.Next(1, 9999)} {streets[rand.Next(streets.Length)]}, " +
               $"{cities[rand.Next(cities.Length)]},
{states[rand.Next(states.Length)]} " +
               $"{rand.Next(10000, 99999)}";
    }

    // Dictionary to map (Airport1, Airport2) -> Flight Time in Minutes
    public static Dictionary<(int, int), int> flightTimes = new()
```